510.84
Il6r
no.350

math

Report No. 350

THE AUTOMATIC GENERATION OF FLOYD PRODUCTION SYNTACTIC ANALYZERS

Alan J. Beals
Jacques E. LaFrance
Robert S. Northcote

September 9, 1969

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

THE AUTOMATIC GENERATION OF FLOYD PRODUCTION SYNTACTIC ANALYZERS*

by

Alan J. Beals
Jacques E. LaFrance
Robert S. Northcote

* This work was supported in part by the Department of Computer Science, University of Illinois, Urbana, Illinois, and in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. USAF 30(602)4144.

ABSTRACT

This paper describes an algorithm for the conversion of a grammar in the form of a set of BNF productions into a deterministic parsing algorithm as described by a set of modified Floyd productions. It describes the implementation of a recognizer based on Floyd productions, including optimization of the recognizer and syntactic error recovery. A complete example is given in an appendix and illustrations from it are used in the text.

Computing Reviews Category Numbers: 4.1 and 4.2

1. Introduction

Floyd Production Language (FPL), developed by Floyd [1] and later modified by Evans [2] and Feldman [3], has often been used to specify the syntax of a language for writing compilers. An FPL syntax specification of a language $\mathcal{L}$ is a direct specification of a (usually) very fast deterministic parsing algorithm for $\mathcal{L}$.

The specification is written as a set of labeled groups of FPL statements (often called "productions", although "reductions" would be more appropriate). Many minor variations on the form of the statements and on the mnemonic meanings of the labels have been used. The form used in this paper is influenced by the fact that the statements, and the labels to them, are generated automatically. The notation used will be described as it is introduced. An FPL statement will have the form:

$$\begin{bmatrix} L_1: \\ \lambda \end{bmatrix} \quad \alpha | \begin{bmatrix} \beta \\ \lambda \end{bmatrix} \qquad \begin{bmatrix} \rightarrow N \\ \leftarrow \bar{N} \\ \lambda \end{bmatrix} | \qquad \begin{bmatrix} * \\ \lambda \end{bmatrix} L_2$$

where

(a) L_1: (optional) labels the group L_1 of 1 or more FPL statements;

(b) α is a string of n symbols which is to be compared with the top n symbols of the recognition stack;

(c) β (optional) is a string of m symbols to be compared with the next m symbols on the input string;

(d) $\rightarrow N$ (optional) means reduce the string α in the recognition stack to the single nonterminal symbol N;

(e) $\leftarrow\overline{N}$ (optional) means place the marker symbol $\overline{N}$ on the marker stack to indicate that the nonterminal symbol N is being sought;

(f) $*$ (optional) is a call on the lexical analyzer (scanner) to place the next symbol from the input string into the recognition stack;

(g) L_2 is the label of the next group of statements to be executed;

(h) if the comparisons in (b) and (c) are unsuccessful, the next statement in sequence (in the same group) is executed;

(i) if there are no statements left in the group, a syntactic error has occurred.

An example of a 36 statement FPL syntax specification of a simple language is given in the Appendix.

Most FPL specifications have been hand coded. While this is not a particularly difficult task the syntax definition so obtained is of little benefit to programmers who wish to program in $\mathcal{L}$. They would prefer to work from a BNF specification of $\mathcal{L}$. Earley [4] and DeRemer [5] have proposed schemes to convert BNF productions into FPL statements. The algorithm described in this paper is an extension and implementation of DeRemer's algorithm.

The use of special markers (bar symbols) in an auxiliary stack facilitates the automatic generation of syntactic error recovery, as well as minimizing the number of FPL statements which must be matched. Also, reductions to nonterminal symbols are allowed only in the top of the recognition stack, thereby further reducing the number of stack comparisons required. The algorithm described below enables very fast and efficient recognizers for many BNF grammars to be generated automatically. It has been used successfully to generate parsers for TRANQUIL [6], a language for specifying array-type algorithms, as well as several other languages being developed on the ILLIAC IV project.

2. The Basic Algorithm

Consider a language $\mathcal{L}$ to be defined by a grammar

$$G = (V_T, V_N, S, P),$$

where

V_T = the set of terminal symbols of $\mathcal{L}$ (represented by lower case Latin letters and the system supplied end marker $\perp$);

V_N = a set of nonterminal symbols (represented by upper case Latin letters);

$S \in V_N$ is the objective symbol;

P is a numbered set of BNF production rules defining $\mathcal{L}$ with $\Sigma ::= \perp S \perp$.

Appendix B is a sample BNF grammar with both intermediate and final results of the application of the algorithm. The examples used in this and the following chapters are chosen from this appendix. Some of the examples used will differ slightly from the appendix but in each case the difference will be explained as further details of the algorithm are described.

Define $X \in V_T \cup V_N$ to be a head symbol of $N_1 \in V_N$ if there exist BNF productions:

$$\begin{aligned} N_1 &::= N_2 \ldots \\ N_2 &::= N_3 \ldots \\ &\vdots \\ N_n &::= X \ldots \end{aligned}$$

for $n \geq 1$.

The following formal rules for converting from BNF productions to FPL statements, for determining the formal labels of groups of FPL statements, and for determining the statements which should constitute each group were developed by DeRemer [5].

If α represents the string made up of the first n symbols on the right of BNF production π, then the following BNF to FPL mapping rules apply:

	BNF Production	FPL Statement	
a)	$M ::= \alpha N \ldots$	$\alpha\vert$	$*Nh$
b)	$M ::= \alpha t \ldots$	$\alpha\vert$	$*t(\pi,\ n+1)$
c)	$M ::= \alpha$	$\alpha\vert\rightarrow M\vert$	Mt

In these FPL statements the string α is to be compared with the top n symbols in the recognition stack; * denotes scan another symbol from the source program into the stack; $\rightarrow M|$ means replace the string α in the stack by the nonterminal symbol M. The symbols on the right are labels of the next group of FPL statements to be executed after complete execution of those productions; Nh identifies a group of statements which attempts to locate an initial (<u>h</u>ead) symbol of <u>N</u> in the stack; $t(\pi,\ n+1)$ labels the statement group which attempts to find a terminal t in the top of the stack corresponding to the t in position n+1 of BNF production number π, Mt labels the group which attempts to match constructs with an <u>M</u> at the <u>t</u>op of the stack. For example, BNF production 1

$$\Sigma :: = \perp S \perp$$

converts to FPL productions 1, 14 and 15:

```
   ⊥ |          *Nh-S
  ⊥S |          *t(1, 3)
 ⊥S⊥ |     →Σ|   Nt-Σ
```

for n = 1, 2 and 3, respectively.

Define $X_n(\pi)$ to be the nth symbol on the right side of BNF production π. Then the following rules determine which labels (groups of FPL statements) must exist.

(a) For each $N \in V_N$:

label Nh exists if $\exists\ \pi, n\colon N = X_n(\pi),\ n > 1$

e.g., for S but not for D in the example.

(b) For each $N \in V_N$:

label Nt exists if $\exists\ \pi, n\colon N = X_n(\pi),\ n > 0$

i.e., for all elements of V_N except Σ in the example.

(c) For each $t \in V_t$:

label $t(\pi, n)$ exists for each $\pi, n\colon t = X_n(\pi),\ n > 1$

e.g., for x but not d in BNF production 7

D ::= dEx

The next step is to determine the set of FPL descriptors for each FPL group which must exist, where each descriptor will correspond to (normally) an FPL production. The three rules are:

(a) $D_{Nh} = \{(\pi, 1) | X_1(\pi) \in V_T$ and the left side of BNF production π is a head symbol of N$\}$

e. g., $D_{Nh\text{-}E} = \{(11, 1), (12, 1), (13, 1)\}$.

(b) $D_{Nt} = \{(\pi, n) \mid N = X_n(\pi), n > 0, \text{all } \pi\}$

e.g., $D_{Nt-C} = \{(3, 2), (6, 1)\}$

(c) $D_{t(\pi, n)} = \{(\pi, n)\}$

e.g., $D_{t(1, 3)} = \{(1, 3)\}$

By determining the group labels, then the descriptors, and applying the mapping rules to the set of BNF productions, it is always possible to obtain an equivalent set of FPL statements which may be used as a syntax recognizer for the language specified by the BNF grammar. Unfortunately, this recognizer will usually be nondeterministic. To make it deterministic, FPL statements within a group may have to be reordered, and, when two statements mutually preclude each other (the placement of either before the other will always preclude execution of the second one), expansion of syntactic context in one or both of them may be necessary. In theory this can always be done, but in practice it is necessary to obtain determinism in a way which minimizes the expansion of context.

3. Rules to Make the Algorithm Deterministic

A simple example of mutual preclusion is obtained when the BNF grammar has productions of the form:

```
1:  N ::= ab...
2:  N ::= ac...
```

which lead to the creation in the Nh group of the FPL statements:

```
a|                *t(1, 2)
a|                *t(2, 2)
```

i.e., statements which have identical stack comparison strings,

```
e.g.,  Nh-E:   f|      → E|     Nt-E
               f|               *t(12, 2)
               f|               *t(13, 2)
```

A first attempt at resolution of this problem is to combine the destination group labels of the offending statements into a single combined group label, combine the groups into a single group with that label, and replace the FPL statements by a single statement. No combination of destination groups is possible when one of the precluding statements involved is of the type

$$\alpha| \quad \rightarrow N \;| \quad Nt$$

since reductions are not delayed. In this case an expansion of context will be necessary.

The following revisions are made to the BNF grammar prior to conversion to FPL statements:

(a) Two BNF productions of the form

$$A ::= \alpha B\ldots$$

and $C ::= \alpha t\ldots$

are changed to

$$A ::= \alpha B\ldots$$

$$C ::= \alpha D\ldots$$

$$D ::= t$$

(b) Two BNF productions of the form

$$A ::= \alpha N\ldots$$

and $B ::= \alpha M\ldots$,

where M is a head symbol of N, are changed to

$$A ::= \alpha N$$

$$B ::= \alpha D$$

$$D ::= M$$

e.g., BNF production 3

$$S ::= aCqr$$

becomes BNF productions 3 and 18

$$S ::= aZqr$$

$$Z ::= C$$

since the C of production 3 is a head symbol of the B of production 2.

If the above revisions have been made to the BNF grammar, and neither of two precluding FPL statements involves a reduction, then the statements will be of the form:

$\alpha|$ *Nh

and $\alpha|$ *Mh

or

$\alpha|$ *t(π_1, m)

and $\alpha|$ *t(π_2, m)

where m = n + 1. These can be combined into

$\alpha|$ *ch(p)

or

$\alpha|$ *ct(p)

where the ch(p) and ct(p) groups of statements are the union of the Nh and Mh groups, and the t(π_1, m) and t(π_2, m) groups, respectively, and p is described below.

e.g., FPL production 2

Nh-S a| *ch(1)

is a result of combining

a| *Nh-B

a| *Nh-Z

which arise from the descriptors (2, 1) and (3, 1) for Nh-S.

When a production of the form:

$\alpha|$ *Nh

corresponding to the descriptor (π, n) is executed a special marker (bar symbol) denoted by $\overline{N}(\pi, m)$ is pushed into a separate bar symbol stack. This is represented in the appropriate FPL statement by:

$$\alpha|\leftarrow\overline{N}(\pi, m)|{*}Nh$$

If the statements, arising from descriptors (π_1, n) and (π_2, n) are identical then a special bar symbol $\overline{N}*(p)$ is used in a single combined statement, where p is a pointer to the bar symbols $\overline{N}(\pi_1, m)$ and $\overline{N}(\pi_2, m)$ (there may be more than two). A combined statement involving several different nonterminal symbols is represented by

$$\alpha|\leftarrow(\overline{p})\ |\ {*}ch(p),$$

where p is a pointer to a list of the bar and special bar symbols involved,

e.g., FPL productions 1, 2 and 26

$$\begin{array}{l|l|l} \perp & \leftarrow\overline{S}(1, 2) & {*}Nh\text{-}S \\ a & \leftarrow(\overline{1}) & {*}ch(1) \\ d & \leftarrow\overline{E}*(2) & {*}Nh\text{-}E \end{array}$$

The FPL statements in each Nt group are classified as type a or type b, according as their descriptors (π, n) have n = 1 or n > 1, respectively. Only type a statements are relevant in the syntax analysis if the symbol at the top of the bar symbol stack is not $\overline{N}$, or a pointer (p) to a symbol list containing $\overline{N}$, because a terminal head symbol is always sought first to begin the construction of a nonterminal and a bar symbol is not, therefore, pushed onto the bar stack for any nonterminal head symbol, i.e., the latter is never explicitly looked for. Thus the type a FPL statements form a subgroup, labeled Nta. The Nta subgroup is void if N is never a head symbol of a BNF definition of a nonterminal other than itself;

e.g., Nta-C: $C|j \qquad {*}t(6, 2)$

$C|\ \rightarrow Z\ |\ Nt\text{-}Z$

If, after a reduction to N, the top symbol in the bar stack is $\overline{N}(\pi, n)$ or a pointer (p) to a symbol list containing $\overline{N}(\pi, n)$ then either a left recursive production (to continue reduction to N) or the FPL statement with descriptor (π, n) applies next. Thus an Ntb(π, n) subgroup which includes the FPL statements arising from the recursive BNF production defining N, followed by a statement to remove the top symbol from the bar stack, followed by the statement with descriptor (π, n) is generated. If the top bar symbol is $\overline{N}*(p)$, or if this symbol is included in a combined list at the top of the stack, then a combined subgroup cNtb(p) is generated in like manner but with the several statements determined by the descriptors in the list pointed to by (p) being placed after the bar removal statement;

```
e.g., Ntb(9, 2):    F|           *t(10, 2)
                    pop bar stack
                   iF|    →D|    Nt-D
```

The combination rules described above may now be applied in a Nta subgroup, in the recursive part of a Ntb or cNtb subgroup, and in the nonrecursive part of a cNtb subgroup. No combination is allowed between recursive and nonrecursive statements in any subgroup.

With this subgrouping, transfer to the group label Nt becomes a dynamic transfer, DNt, to Nta, a Ntb(π, n) or a cNtb(p) subgroup depending on the DNt symbol currently at the top of the bar stack;

```
e.g., FPL production 21
                   iF|    →D|    DNt-D
```

If preclusions still exist after the subgrouping and combining described above then contextual expansion is required. A great deal of information concerning the symbols preceding the string α in the stack comparison part of an FPL statement is implicit in the grouping of the statement. Therefore, only right context expansion (lookahead) is employed. This is done by generating, for each statement involved in the preclusion, all possible strings (up to length k) of terminal symbols which may follow the α in the stack comparison part. If all such strings are different for the involved statements the preclusion has been eliminated. Experience has shown that a lookahead of k = 1 symbol is usually sufficient to eliminate preclusions and no practical examples have been found where a finite look-ahead of more than k = 3 symbols is necessary to resolve a preclusion. Thus, when contextual analysis is necessary, a one symbol lookahead is generated. If this fails to differentiate, a three symbol lookahead is generated. If this also fails to resolve preclusions the attempt to obtain a deterministic FPL recognizer from the given BNF grammar is terminated. The lookahead for a particular FPL production need only be enough to differentiate it from any following productions which it precludes. Thus the last of a set of precluding productions needs no lookahead;

```
e.g., Nh-E:   f|x    →E|   DNt-E
               |y
              f|           *ct(4)
```

4. Optimization of Interpretive Instructions

The construction of a syntactic analyzer out of the FPL statements involves the construction of a string of operators and operands which can be either interpretively executed or converted to an ALGOL program to be compiled and then executed directly. The operators fall into seven classes: pointer initializing, recognition stack tests, lookahead queue tests, recognition stack manipulation, transfer of control, semantic routine calls, and error recovery, a complete list of which is given in Appendix A.

Recognition Stack Tests

In the Nta, Ntb(π, n) and cNtb(p) type subgroups no recognition stack tests are necessary. In the Nta type subgroups, such as Floyd production number 12:

```
        Nta-D:    D|              *  ct(3)
```

the string α consists of the single symbol D which is put there immediately before transfer to this group either by production number 21:

```
                  iF|      →D | DNt-D
```

or by production number 28:

```
                  i|       →D | DNt-D
```

In the Ntb(π, n) and cNtb(p) type groups the string α is of the form βN, where β is of length n, n $\geq$ 1, for example, Floyd production number 21 above. In this case, the presence of the string β = "i" is verified just before the symbol $\overline{N}(\pi, n)$(= "$\overline{F}$(9, 2)") is pushed into the bar stack, in production number 27:

$$i \left| \begin{array}{l} m \\ k \\ \ell \end{array} \right. \qquad \leftarrow\overline{F}(9,\ 2) \ \Big| \quad *\ \text{Nh-F}$$

and a transfer is made to the Nh-F group to begin seeking the constituents of F. When F has been found, the recognition stack contains iF, the top symbol in the bar stack is then $\overline{F}(9, 2)$, and control is transferred to the type b subgroup, Ntb(9, 2).

In the Nh, and ch(p) type groups, a terminal head symbol of a non-terminal is sought. Hence, the α's all consist of single terminal symbols, as in Nh-F:

k |	$\rightarrow$F	| DNt-F
ℓ |	$\rightarrow$F	| DNt-F
m |	$\rightarrow$F	| DNt-F

In the ct(p) type groups the α in the stack is of the form βt where the β was recognized in the previous production and the t is the symbol scanned just before the transfer to this group. For example, ct(5):

dEx |	$\rightarrow$C	| DNt-C
dEy |	$\rightarrow$C	| DNt-C

where the "dE" has been recognized in production 30:

dE |		*ch(5)

Hence, in all three of these type groups it is sufficient to test only the top (terminal) symbol of the recognition stack.

The statements in each of these groups are ordered before they are processed so that all those with identical stack comparisons (differentiated by lookahead) are together. The symbol at the top of the recognition stack is tested on the first statement with an instruction which, upon failure, transfers to the beginning of the next statement with a different symbol at the top of the stack. No stack test is made on the statements in between. An example is group ch(1):

```
d |           ←Ē*(2)  |   * Nh-E

  | m
i | k         ←F̄(9, 2)|   * Nh-F
  | ℓ

i |           →D      |     DNt-D
```

In this case the test for "i" in the third production is not made. Failure in the stack test in the second production causes a transfer over the third production to the end of the group; failure in the lookahead test in the second production causes a transfer to the next (third) production. In the event that several statements have different stack comparisons but all take the same actions (same semantic routine calls and same recognition stack reduction), a mode pattern is built with a bit on for the stack symbol of each statement. One instruction is produced for the stack tests of all of these statements. It simply checks to see if, in the appropriate row, the bit corresponding to the top stack symbol is on. The Nh-F group mentioned above is an example of this.

Since the t(π, m) groups are identical in basic form to the ct(p) groups they are handled in the same way as the preceding except that there is always only one statement in each group and only one transfer to each group (which yields a further optimization to be discussed later).

Lookahead Contextual Analysis Tests

These tests follow the recognition stack tests whenever needed and are implemented with three main types of instructions:

(1) if the right symbol is present, increment the lookahead level pointer and go on with next instruction, otherwise branch to another instruction;

(2) if the right symbol is present, branch to another instruction, otherwise go on with the next instruction;

(3) if the right symbol is present, go on with next instruction, otherwise branch to the beginning of the next statement.

The address of the next statement is set in a global location just before the string of lookahead test instructions. Each of the last two types include a bit pattern test like that used in the stack tests above. To optimize the lookahead test the lookahead strings are ordered as in the following example:

given strings	ordered
cef	a
cd	b
cij	cd
cik	cef
a	ceg
ceg	ceh
b	cij
cil	cik
ceh	cil

The following figure shows how the above lookahead strings would be marked for testing with subscripts indicating which of the three types of test is needed for each symbol. Subscript 0 indicates that no test is needed, and x, y and z are the bit patterns formed for this example:

$$\begin{array}{lll}
\left.\begin{array}{l}a_2\\b_2\end{array}\right] & x & \\
c_3 & d_2 & \\
\begin{array}{l}c_0\\c_0\\c_0\end{array} & \begin{array}{l}e_1\\e_0\\e_0\end{array} & \left.\begin{array}{l}f_2\\g_2\\h_3\end{array}\right] \; y \\
\begin{array}{l}c_0\\c_0\\c_0\end{array} & \begin{array}{l}i_3\\i_0\\i_0\end{array} & \left.\begin{array}{l}j_2\\k_2\\\ell_3\end{array}\right] \; z
\end{array}$$

$a, b \in x$

$f, g, h \in y$

$j, k, l \in z$

Let XLB be the instruction type 1 (branch on failure), XLA be type 2 (branch on success), and XLL be type 3 (branch to next production on failure), with a suffix "B" meaning a bit-pattern test, and L_i representing a transfer to label L_i as indicated by the instruction type. Then the following is the list of instructions performing this lookahead contextual analysis:

```
     XLAB (x, L2)
     XLL  (c)
     XLA  (d, L2)
     XLB  (e, L1)
     XLLB (y)
     GOTO (L2)
L1:  XLL  (i)
     XLLB (z)
L2:  {rest of this Floyd production}
```

Transfer of Control

Some additional optimization can be applied if the symbol following α is a terminal which is not combined. The fact indicated earlier, namely, that there is always only one statement in a $t(\pi, m)$ group and only one transfer to it, means that there is no need to generate a transfer. Therefore, the next symbol is scanned and the $t(\pi, m)$ statement is created immediately as if it were part of the same statement. Floyd productions 18 and 19 are an example:

```
                  F | n                  *     t(10, 2)
          t(10, 2):         Fn |        →F  |             DNt-F
```

Further, if a lookahead was performed then the symbol was successfully tested in the lookahead test so no test need be performed in the $t(\pi, m)$ statement. The next symbol is scanned and the recognition stack pointer is incremented without a test. This is the case in the above example. If a lookahead of at least k symbols is required, then this is done for $t(\pi, n + i)$ $(1 \leq i \leq k)$ statements, if they exist.

Several other minor optimizations involving stack reduction and transfer of control have been implemented, as indicated by the description of the parser operators given in Appendix A.

5. Error Recovery

At the end of each group there is an error statement which is applied if every statement in that group failed to match. The general error recovery technique used has been to look at the top symbol in the bar symbol stack, reduce the stack to the nonterminal named in the case of an $\overline{N}$ or $\overline{N}*$, skip the input to the first symbol which can follow N, and transfer to the Ntb group indicated by the bar symbol. If the bar symbol is a combined bar symbol, then the input is scanned for the first symbol that can follow any of the bar symbols in the combined group. When one is found the bar symbol it follows is treated as above. This implies the existence of a table which gives, for each occurrence of a nonterminal, a table of terminal symbols which may immediately follow that nonterminal occurrence. This table actually includes terminal occurrences also, as will be seen later, and is generated in the same manner as the lookahead strings are generated.

It sometimes happens that the symbol in error is first encountered in a lookahead test. This can cause the appropriate FPL statement to be skipped in favor of a wrong one. The parse is directed down a wrong path and several reductions sometimes can be made and bar symbols popped before the error is detected. This causes a greater portion of the input to be skipped than if all the bar symbols had been retained.

An ALGOL example of this problem would be the following: Suppose an arithmetic expression in an assignment statement contains an incorrect exponentiation operator. Then when it becomes the next input symbol, the bar stack will contain $\overline{\text{primary}}$, $\overline{\text{factor}}$, $\overline{\text{term}}$, $\overline{\text{arithmetic expression}}$, $\overline{\text{statement}}$, $\overline{\text{compound tail}}$, $\overline{\text{program}}$. The lookaheads needed to decide if the end of each construct has been reached could be ordered in such a way that the reduction is always made and the bar symbol popped until the END following the last statement is needed, and inserted, to make the compound tail until, finally, only the $\overline{\text{program}}$ symbol is left. Then the input is scanned to the first symbol following program, namely end-of-file mark, i.e., the rest of the source string is skipped; whereas, if the primary were the top bar symbol only a few symbols to the next operator, "; ", END, ELSE, etc. would have been skipped.

In order to avoid this problem an additional test is made in any group where lookahead testing is needed to check whether the next symbol is in the set of symbols which can occur at that point. If it cannot, then the general error recovery scheme is called immediately.

There are many situations which have special features and allow for more specialized error recovery than that outlined above. A discussion of these now follows.

The $t(\pi, m)$ type of statement is executed when the previous history of the parse leaves no choice. Since there is only one production and, therefore, only one possibility for the top symbol in the stack it may be inserted by the following insertion rules if it is not there. The parse then may proceed as if no error had occurred; no error production is required.

Rules for Insertion

Assume the following symbolism: β = the part of the stack below the top symbol, γ = the unscanned portion of the input after the next symbol, a = the symbol that is sought by the test, b = any symbol which can immediately follow this occurrence of a, c = any symbol, $|$ = the top of the stack (the stack to the left, and the unscanned input to the right, $\Longrightarrow$ = if the situation to the left of the arrow holds, then change it to the situation on the right. The following are the rules; the first one from the top that applies is the one that is used:

$$\beta\ b \mid a\ \gamma \Longrightarrow \beta\ a \mid b\ \gamma$$
$$\beta\ b \mid c\ \gamma \Longrightarrow \beta\ a \mid b\ c\ \gamma$$
$$\beta\ c \mid a\ \gamma \Longrightarrow \beta\ a \mid \gamma$$
$$\beta\ c_1 \mid c_2\gamma \Longrightarrow \beta\ a \mid c_2\gamma$$

The second special case occurs when an Nh, ch(p), or ct(p) group consists only of statements all of which contain the same symbol in the stack comparison field; then the stack test can insert the symbol if it is not there, using the above rules for insertion as in the $t(\pi, m)$ statements. The error statement is not then needed since the parse will continue in the same way regardless of the outcome of the stack comparison.

The next special case is that of a group in which one statement has a stack symbol of a character or special word whereas all the others of that group have, as stack comparison symbol, a terminal class symbol (identifier, number, or string). In this case the assumption is made that the error is far more likely to have occurred with the specific terminal symbol than with a terminal class symbol. The error statement here inserts the specific terminal symbol according to the rules for insertion and transfers back to the appropriate place in the corresponding statement.

The last special case applies when all the statements of a group make the same reduction. In this case that reduction is made anyway. The top symbol of the stack, which didn't match any of the stack tests, is put back into the input queue if it can follow the nonterminal to which the stack is reduced. The Nh-F group is an example:

k	→F	DNt-F
ℓ	→F	DNt-F
m	→F	DNt-F

Note that no error statement is needed in the Nta, Ntb(π, m) and cNtb(p) type subgroups because no stack test is made in these groups and the last production requires no lookahead test so it, at least, will match.

6. Conclusion

The FPL parsing algorithm, not surprisingly, has similarities to precedence parsing algorithms in that the three different possible stack actions do nothing, $\leftarrow\overline{N}$, $\rightarrow N$, which can be specified in a FPL statement, correspond to the three precedence operators $\doteq$, $\lessdot$, and $\gtrdot$, respectively. The conversion algorithm is rather slow compared with some other algorithms but has the advantage that, with one exception, it is able to make use of more context than is employed in precedence schemes in determining the bounds of the phrase next to be reduced. A more general error recovery capability than that usually associated with precedence techniques is included in the algorithm.

Careful consideration of rather obvious optimizations in the information included in the FPL statements has been reflected in the FPL statement generation algorithm, thereby enabling the production of highly efficient syntax recognizers. Representation of the basic parser interpreter instructions as hardware instructions, or at least as microprogrammed sequences, would further enhance overall compiler performance. Interpretation of syntax tables also can be avoided by directly coding the FPL statements in a higher level language for machines, such as the B5500 and B6500, which have a suitably matched software-hardware capability.

Appendix A: Parser Operators

Mnemonic	Operands	Action
LLVL		Initialize lookahead buffer test level pointer to the first position
ILVL		Increment the recognition stack pointer

XSBS	S,A	Test the top of the stack for symbol S
		yes => increment stack pointer
		no => branch to address A

XSBT	T,A	Test the top of the stack for class symbol type T
		yes => increment stack pointer
		no => branch to address A

XSBB	R,A	Test the top of the stack with row R of the pattern array marked => increment stack pointer not marked => branch to address A
XSIS	S	Test the top of the stack for symbol S yes => increment stack pointer no => insert S at top of stack and increment stack pointer
XSIT	T	Test the top of the stack for class symbol type T yes => increment stack pointer no => insert a symbol of type T at top of stack and increment stack pointer
XLBS	S,A	Test the input queue for symbol S yes => increment lookahead level pointer no => branch to address A

Mnemonic	Operands	Action
XLBT	T,A	Test the input queue for class symbol type T yes => increment lookahead level pointer no => branch to address A

XLAS	S,A	Test the input queue for symbol S
		yes => branch to address A
		no => go on

XLAT	T,A	Test the input queue for class symbol type T
		yes => branch to address A
		no => go on

XLAB	R,A	Test the input queue with row R of the pattern array marked => branch to address A not marked => go on
XLLS	S	Test the input queue for symbol S yes => go on no => branch to address in NEXTPRØDUCTIØN
XLLT	T	Test the input queue for class symbol type T yes => go on no => branch to address in NEXTPRØDUCTIØN
XLLB	R	Test the input queue with row R of the pattern array marked => go on not marked => branch to address in NEXTPRØDUCTIØN
NPØP	N	Subtract N from the recognition stack pointer
RED1	S	Change the name of the top symbol of the recognition stack to S

Mnemonic	Operands	Action
REDN	N,S	Subtract N from the recognition stack pointer and change the name of the top symbol of the recognition stack to S
BPSH	B	Push bar symbol B into the bar stack
BPØP		Pop the top bar symbol from the bar stack
TPSH		Put the next input symbol into the recognition stack at location of the recognition stack pointer
EXEC	N	Execute semantic routine N
XTSM	N	Execute semantic routine N and test global Boolean SEMANTICTEST true => go on false => branch to address in NEXTPRØDUCTIØN
XRSM	N	Execute semantic routine N and test global Boolean SEMANTICTEST true => go on false => print error message and go on
NØØP		Go on
SKIP	N	Skip N characters to next row of parser instruction table
GØTØ	A	Branch to address A
XBGØ	A	Test top stack symbol with top bar stack symbol (possibly going into a combined group) match => branch to address in bar symbol no match => branch to address A
SETS	A	Put A in NEXTPRØDUCTIØN

Mnemonic	Operands	Action

XSLR		Test top of stack with next input symbol to see if latter can follow the former
		yes => go on
		no => execute code for ERRR instruction

ERRT	S,A	Print error message, insert terminal symbol S at top of stack and go to address A
ERRN	S,A,	Print error message, reduce stack to nonterminal symbol S, and go to A
ERRR		Print error message, recover from error by using top bar symbol

Appendix B: Conversion of a Simple BNF Grammar

(a) The BNF productions:

Production number	1:	Σ	::=	$\perp$ S $\perp$	
	2:	S	::=	a B	|
	3:			a C q r	
	4:	B	::=	D b	|
	5:			D c	|
	6:			C j	
	7:	C	::=	d E x	|
	8:			d E y	
	9:	D	::=	i F	|
	10:			i	
	11:	E	::=	f	|
	12:			f g	|
	13:			f h	
	14:	F	::=	F n	|
	15:			k	|
	16:			ℓ	|
	17:			m	

A dummy nonterminal symbol Z is needed at (3,2) so production 3 is changed and 18 is added as follows:

3:	S	::=	a Z q r
18:	Z	::=	C

(b) FPL statement group labels and descriptors:

FPL Statement Group Label	Descriptor Set
Nh-Σ (start)	(1,1)
Nh-S	(2,1)(3,1)
Nh-E	(11,1)(12,1)(13,1)
Nh-F	(15,1)(16,1)(17,1)
Nta-Σ	exit
Nta-C	(6,1)(18,1)
Nta-D	(4,1)(5,1)
Ntb(1,2)	(1,2)
Ntb(2,2)	(2,2)
Ntb(3,2)	(3,2)
Ntb(9,2)	(14,1)(9,2)
ch(1)	(7,1)(8,1)(9,1)(10,1)
cNtb(2)	(7,2)(8,2)
ct(3)	(4,2)(5,2)
ct(4)	(12,2)(13,2)
ct(5)	(7,3)(8,3)
t(1,3)	(1,3)
t(3,3)	(3,3)
t(3,4)	(3,4)
t(6,2)	(6,2)
t(14,2)	(14,2)

(c) The FPL statements generated:

```
1.  Nh-Σ (start):    ⊥|          ← S̄(1,2)|   *    Nh-S

2.  Nh-S:            a|          ← (̅1̅)̅   |   *    ch(1)

                     f|x         → E      |        DNt-E
3.  Nh-E:             |y
4.                   f|                       *    ct(4)

5.  Nh-F:            k|          → F      |        DNt-F

6.                   ℓ|          → F      |        DNt-F

7.                   m|          → F      |        DNt-F

8.  Nta-Σ:           success exit

9.  Nta-C:           C|j                      *    t(6,2)

10.             t(6,2):      Cj|      → B|              DNt-B

11.                  C|          → Z      |        DNt-Z

12. Nta-D:           D|                       *    ct(3)

13. Ntb(1,2):              pop bar stack

14.                  ⊥S|                      *    t(1,3)

15.             t(1,3):     ⊥S⊥|      → Σ|              DNt-Σ

16. Ntb(2,2):              pop bar stack

17.                 aB|          → S      |        DNt-S

18. Ntb(9,2):        F|n                      *    t(10,2)

19.             t(10,2):    Fn|       → F|              DNt-F

20.                        pop bar stack

21.                 iF|          → D      |        DNt-D
```

22.	Ntb(3,2):	pop bar stack			
23.		aZ|		*	t(3,3)
24.	t(3,3):	aZq|		*	t(3,4)
25.	t(3,4):	aZqr|	→ S|		DNt-S
26.	ch(1):	d|	← $\overline{E}^{*}(2)$ |	*	Nh-E
27.		i| m k ℓ	← $\overline{F}(9,2)$|	*	Nh-F
28.		i|	→ D |		DNt-D
29.	cNtb(2):	pop bar stack			
30.		dE|		*	ch(5)
31.	ct(3):	Db|	→ B |		DNt-B
32.		Dc|	→ B |		DNt-B
33.	ct(4):	fg|	→ E |		DNt-E
34.		fh|	→ E |		DNt-E
35.	ct(5):	dEx|	→ C |		DNt-C
36.		dEy|	→ C |		DNt-C

(d) The FPL parser interpreter instructions

```
Nh-Σ:            push ⊥ into recognition stack
                 TPSH
                 GOTO (L1)
         L0:     ERRR
Nh-S:    L1:     XSIS (a)
                 BPSH (1̄)
                 TPSH
                 GOTO (L16)
Nh-E:    L2:     XSIS (f)
                 SETS (L3)
                 LLVL
                 XLLB (1)        row 1 of pattern array:  x,y
                 RED1 (E)
                 XBGO (L0)
         L3:     TPSH
                 GOTO (L24)
Nh-F:    L4:     XSBB (2, L6)    row 2 of pattern array:  k,l,m
         L5:     RED1 (F)
                 XBGO (L0)
         L6:     ILVL
                 ERRN (F, L5)
Nta-Σ:   L7:     success exit
Nta-C:   L8:     SETS (L9)
                 XSLR
                 LLVL
                 XLLS (j)
                 TPSH
(t(6,2):)        ILVL
                 REDN (1, B)
                 XBGO (L0)
         L9:     RED1 (Z)
                 XBGO (L0)
Nta-B:   L10:    TPSH
                 GOTO (L21)
```

```
Ntb(1,2): L11:   BPOP
                 TPSH
                 XSIS (⊥)
                 REDN (2, Σ)
                 XBGO (L7)
Ntb(2,2): L12:   BPOP
                 REDN (1,S)
                 XBGO (L0)
Ntb(9,2): L13:   SETS (L14)
                 XSLR
                 LLVL
                 XLLS (n)
                 TPSH
(t(10,2):)       ILVL
                 NPOP (1)
                 XBGO (L0)
          L14:   BPOP
                 REDN (1, D)
                 XBGO (L10)
Ntb(3,2): L15:   BPOP
                 TPSH
(t(3,3):)        XSIS (q)
                 TPSH
(t(3,4):)        XSIS (r)
                 REDN (3, B)
                 XBGO (L0)
ch(1):    L16:   XSBS (d, L17)
                 BPSH (E⃗(2))
                 TPSH
                 GOTO (L2)
          L17:   XSBS (i, L19)
                 SETS (L18)
                 LLVL
                 XLLS (2)
                 BPSH (F̄(9,2))
                 TPSH
                 GOTO (L4)
```

```
            L18:     RED1 (D)
                     XBGO (L10)
            L19:     ILVL
                     ERRR
cNtb(2):    L20:     BPOP
                     TPSH
                     GOTO (L27)
ct(3):      L21:     XSBB (3, L23)    row 3 of pattern array:  b,c
            L22:     REDN (1, B)
                     XBGO (L0)
            L23:     ILVL
                     ERRN (B, L22)
ct(4):      L24:     XSBB (4, L26)    row 4 of pattern array:  g,h
            L25:     REDN (1, E)
                     XBGO (L0)
            L26:     ILVL
                     ERRN (E, L25)
ct(5):      L27:     XSBB (1, L29)
            L28:     REDN (2, C)
                     XBGO (L8)
            L29:     ILVL
                     ERRN (C, L28)
```

References

[1] Floyd. R. W. "A descriptive language for symbol manipulation", J. ACM 8 (Oct, 1961), p 579-584.

[2] Evans, A. "An Algol 60 compiler", Annual Review in Automatic Programming, Vol. 4, 1964, p 37-50.

[3] Feldman, J. A. "A formal semantics for computer languages and its application in a compiler-compiler", Comm ACM 9 (Jan, 1966), p 3-9.

[4] Earley, J. C. "Generating a recognizer for a BNF grammar", Report, Computation Center, Carnegie Mellon University (1965).

[5] DeRemer, F. L. "On the generation of parsers for BNF grammars: an algorithm", Proc SJCC (1969).

[6] Abel, N. E., Budnik, P. P., Kuck, D. J., Muraoka, Y., Northcote, R. S., and Wilhelmson, R. B. "TRANQUIL: a language for an array processing computer", Report No. 315, Department of Computer Science, University of Illinois at Urbana-Champaign, (April, 1969).

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

THE AUTOMATIC GENERATION OF FLOYD PRODUCTION SYNTACTIC ANALYZERS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Research Report

5. AUTHOR(S) (First name, middle initial, last name)

Alan J. Beals, Jacques E. LaFrance, and Robert S. Northcote

6. REPORT DATE September 9, 1969	7a. TOTAL NO. OF PAGES 38	7b. NO. OF REFS 6
8a. CONTRACT OR GRANT NO. 46-26-15-305 b. PROJECT NO. US AF 30(602)4144 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) DCS Report No. 350 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies from DCS.

11. SUPPLEMENTARY NOTES NONE	12. SPONSORING MILITARY ACTIVITY Rome Air Development Center Griffiss Air Force Base Rome, New York 13440

13. ABSTRACT

This paper describes an algorithm for the conversion of a grammar in the form of a set of BNF productions into a deterministic parsing algorithm as described by a set of modified Floyd productions. It describes the implementation of a recognizer based on Floyd productions, including optimization of the recognizer and syntactic error recovery. A complete example is given in an appendix and illustrations from it are used in the text.

DD FORM 1 NOV 65 1473

UNCLASSIFIED
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
programming language syntax						
syntactic analysis						
parsing algorithm						
compiler						
error recovery						
Floyd productions						